I must Create a System, or be enslav'd by another Man's.
I will not Reason & Compare; my business is to Create.

—William Blake, *Jerusalem* (1804)

Chapter 2

Algorithm implementations in a software library

Owing to their central importance in mathematical computing, software for polynomial computations exists in numerous programs and libraries that have been developed both in academia and in industry. We will examine this existing software and show that, surprisingly, no open-source high-performance library focusing on polynomial computation — including multivariate and sparse polynomials — is currently available.

We then discuss our proposed solution, a new software library for computations with polynomials, broadly defined, which we call the MVP library. We give an overview of the architecture and aims of the software, and discuss how some low-level operations have been implemented. Although the library is still in its infancy, we demonstrate that it is already competitive in speed with other software, while being at least general enough to hold all the algorithms that will be presented in this thesis.

The author extends his thanks to Clément Pernet for many fruitful discussions, and particular those which started the idea of this software project.

2.1 Existing software

Computations with polynomials are useful in a wide range of applications, from engineering to mathematical research. It would therefore be impossible to discuss every piece of software that includes polynomial computations at some level. Instead, we will attempt a broad overview of existing software that either takes a comprehensive view including such computations as a key component, or that which focuses specifically on some aspect of polynomial computation. Starting at the top level, we review existing and available software, and then highlight the gap in such software that we are attempting to fill.

MACSYMA was first developed starting in 1968 at MIT as the first comprehensive computer algebra system with the general goal of solving all manner of mathematical problems with the

aid of a computer. Since then, numerous such systems have followed suit, albeit with varying aims. The most successful of these today are the commercial products MAPLE, MATHEMATICA, and MAGMA, with open-source programs such as SAGE, MAXIMA, AXIOM, and REDUCE also gaining popularity. Popular comprehensive numerical computation programs such as MATLAB and OCTAVE also have some symbolic computation capabilities, either built-in or available as an add-on package.

These programs have somewhat differing emphases, but they all include algorithms for a wide variety of mathematical problems, from differential equations to linear algebra. Of course also included are computations with polynomials, and these implementations are often highly optimized, as their speed affects the speed of many parts of the system. However, by their general nature as comprehensive systems, these programs all have extensive overhead costs incorporated, and are not well-suited for developing high-performance software for particular applications.

Large but somewhat less comprehensive systems have also been developed for a variety of mathematical problems, and some focus especially on polynomials. COCOA (Abbott and Bigatti, 2011) and SINGULAR (Decker, Greuel, Pfister, and Schönemann, 2010) are comprehensive systems for commutative and non-commutative algebra. They include multivariate polynomial arithmetic and focus specifically on Gröbner basis computations and algebraic geometry. PARI/GP (PAR, 2010) is a program with similar breadth, and also with support for polynomial arithmetic, but with a focus more on number theory. TRIP (Gastineau and Laskar, 2011) is a computer algebra system for celestial mechanics with very efficient sparse multivariate polynomial arithmetic over inexact domains. However, these programs are still quite general, featuring support for a wide range of operations and each with their own programming languages.

Finally, there are a few libraries for low-level computations with polynomials with a narrow focus on high performance for specific problems. A very popular and well-designed example is Victor Shoup's NTL, "a Library for doing Number Theory" (2009). This library contains very efficient implementations for univariate polynomials with integer, rational, and finite field coefficients, as well as basic algebraic operations such as factorization. NTL set the standard for the fastest implementations of these operations, using asymptotically fast algorithms along with simpler ones for small problem sizes, and making use of a variety of low-level tricks and tweaks. However, development has mostly been limited to bug fixes since 2005.

More recently, the FLINT library has emerged as a successor to NTL with similar goals but more active development (Hart, Johansson, and Pancratz, 2011). FLINT's core focus is on efficient arithmetic over $\mathbb{Z}[x]$, but it also has support for other kinds of univariate polynomials and some integer matrix computations as well, including lattice reduction algorithms. David Harvey, one of the original developers of FLINT, has also released an extremely efficient library for arithmetic in $\mathbb{Z}/m\mathbb{Z}$, where m is machine word-sized, called `zn_poly` (2008).

FLINT also contains implementations of multiple-precision integer arithmetic. Although the current work is focused specifically on computations with polynomials, there is an intimate connection with integer computations, as we have already seen. Besides FLINT, the Gnu Multiple Precision library (GMP) by Granlund et al. (2010) is certainly the most popular

library for long integer arithmetic. It contains both C and C++ bindings and has highly-tuned arithmetic routines, often written in assembly for a variety of different architectures.

The author is aware of only one software library focused on high-performance multivariate polynomial arithmetic, the SDMP package by Roman Pearce and Michael Monagan. This package features very efficient algorithms for sparse multiplication and division, and appears to be the fastest for some range of cases, even scaling to multi-cores (Monagan and Pearce, 2009, 2010a,b). However, a major drawback is that the domain is limited mostly to characteristic-zero rings and single-precision exponents. Furthermore, the software is closed-source and is only incorporated into the general-purpose commercial computer algebra system Maple.

Another high-performance library that is only available through Maple is `modpn` (Li, Maza, Rasheed, and Schost, 2009a). This library is mainly designed as the back-end for polynomial system solving via triangular decomposition, and it contains very efficient arithmetic for dense multivariate polynomials.

Finally, there have been a few academic software packages developed for computations on polynomials given in a functional representation. DAGWOOD is a Lisp-based system for computations with straight-line programs made to interface with MACSYMA (Freeman, Imirzian, Kaltofen, and Yagati, 1988). The FOXBOX system allowed computations with polynomials represented by black boxes for their evaluation, including sparse interpolation (Díaz and Kaltofen, 1998). It made use of C++ templates, relied on the use of NTL and GMP, and was built on top of the SACLIB library. Unfortunately, SACLIB appears to no longer be actively supported, and both DAGWOOD and FOXBOX are no longer maintained or even available.

2.2 Goals of the MVP library

Considering the great success of low-level, open-source libraries for dense univariate polynomial arithmetic (namely NTL, FLINT, and `zn_poly`), it is surprising that no such library exists for multivariate and/or sparse polynomials. We are very thankful to the authors of efficient multivariate arithmetic libraries such as TRIP, SDMP, and `modpn` for publishing extensively on the design and implementation of their software. However, since their programs are closed-source, some details will always be hidden, and it will be difficult if not impossible to interface with their libraries, either by writing new algorithms to be inserted, or by including their libraries in a larger system. On the latter point, it should be noted that the open-source licenses and development of FLINT and PARI have led to their wide use as the back-end for polynomial computations in SAGE.

Our goal therefore is a software library that is free and open-source and focuses on dense and sparse multivariate polynomial arithmetic and computations. In design, the software should be flexible enough to compare different algorithms, including interfacing with existing software, while also achieving high performance. In addition, the development should be open to allowing experimentation in new algorithms by researchers in symbolic computation. To our knowledge, no such library currently exists.

One consequence of this gap is that numerous algorithms developed over the past few decades currently have no readily-available implementation for experimentation and comparison. For instance, the celebrated sparse interpolation algorithm of Ben-Or and Tiwari (1988) and the sparse factorization algorithm of Lenstra (1999), while probably used as the back-end in at least a few existing software packages, have no currently-maintained implementations. This hinders both the development and widespread use of sparse polynomial algorithms.

In the remainder of this chapter, we describe the design and implementation of a new software library for polynomial arithmetic, which we call MVP (for MultiVariate Polynomials — also, we hope the library will be most valuable). The general goals are as described above, and the scope of algorithms is the same as those outlined in Chapter 1: arithmetic operations, algebraic computations, and inverse symbolic computation. The current state of the library is very immature and covers only basic arithmetic and the algorithms of this thesis. However, the architecture of the library is designed to allow significant additions in the coming years, including both celebrated past results that are currently without implementations, as well as (hopefully) future advances in algorithms for polynomial computation.

2.3 Library design and architecture

The design of the MVP library borrows heavily from FoxBox as well as the popular exact linear algebra package LinBox (Dumas, Gautier, Giesbrecht, Giorgi, Hovinen, Kaltofen, Saunders, Turner, and Villard, 2002). These packages organize data structures by mathematical significance and make extensive use of C++ templates to allow genericity while providing high performance. This genericity allows numerous external libraries to be "plugged in" and used for underlying arithmetic or other functionality.

We briefly describe some architectural features of the MVP library, followed by an overview of its contents and organization.

2.3.1 Library design

Template programming in C++ is essentially a built-in tool for compile-time code generation. Hence template libraries can achieve the same run-time performance as any C library, but have greater flexibility, composability, and maintainability.

For instance, an algorithm for multiplication of densely-represented univariate polynomials over an arbitrary ring can be written once, then used in any situation. At compile-time, the exact function calls for the underlying ring arithmetic are substituted directly, so that no branches or look-up tables are required at run-time.

Of course, template programming also has its drawbacks. For instance, consider a recursive polynomial representation using templates, so that an n-variate polynomial is defined as a univariate polynomial with $(n-1)$-variate polynomial coefficients. This could lead to

a class definition such as `Poly< Poly< Poly< Poly< Ring > > > >`, which quickly becomes unwieldy and can result in tremendous code bloat. As compilers and hardware continue to improve, the cost of such expansive templates will decrease. However, the strain on users of the library will still be severe, especially considering that they may no be expert programmers but mathematicians requiring moderate but not optimal performance.

For these reasons, the MVP library also allows for polymorphism. This is achieved through abstract base classes for basic types such as polynomial rings, which then have a variety of implementing subclasses. Using polymorphism instead of templates means that types must be checked at run-time. This results in somewhat weaker performance, but potentially smaller code, faster compilation, and much easier debugging. By making full use of templates but also allowing polymorphism, the MVP library gives users more flexibility to trade off ease of programming with performance.

2.3.2 Main modules

The MVP library is divided into six modules: `misc`, `ring`, `poly`, `alg`, `test`, and `bench`. The coupling is also roughly in that order; for instance, almost every part of the library relies on something in the `misc` module, whereas nothing outside of the `bench` module relies on its contents. We now briefly describe the design and current functionality of each module.

`misc`: This module contains common utilities and other functionality that is useful in a variety of ways throughout the package. Functions for error handling and random number generation are two examples. Most functions in this module are not template functions and hence can be pre-compiled.

Handling random number generation globally allows for easily repeatable computations, by re-using the same seed. This is very important not only for benchmarking and debugging, but also for algorithm development and mathematical understanding. As we will see in later chapters of this thesis, the performance of some algorithms relies on unproven number-theoretic conjectures, and a failure may correspond to a counter-example to said conjectures.

`ring`: The abstract class `Ring` is the superclass of all coefficient types in the library. Here we follow the lead of LinBox in separating the ring class from ring elements. It is tempting to write a C++ class for elements, say of the ring $\mathbb{Z}/m\mathbb{Z}$ for small integers m. Operator overloading would then allow easy use of arithmetic in this ring. However, this requires that the modulus m (and any other information on the ring) either be contained or pointed to by each field element, massively wasting storage, or be globally defined. NTL takes the latter approach, and as a result is inherently not able to be parallelized.

Rather, we have a C++ class for the ring itself which contains functions for all kinds of arithmetic in the ring. Algorithms on ring elements must then be passed (or templatized on) the ring type. Elements of the ring can be any built-in or user-defined type, allowing more compact storage. The disadvantage to the user of not being able to use overloaded arithmetic operators is the only drawback.

Besides the abstract base class, the `ring` module also contains implementations for integer modular rings, arbitrary-precision integers and rational numbers, and approximate complex numbers. There are also adaptor classes to link with existing software such as NTL.

`poly`**:** The abstract class `Poly` is a subclass of `Ring` and the superclass of all polynomial ring types. The class is templatized over the coefficient ring, the storage type, and the number of variables. Using the number of variables as a template parameter will allow for highly efficient and easily maintainable recursive types, and specialisations for univariate and (perhaps in the future) bivariate polynomials are easily defined.

Separating the polynomial rings from their storage is not as important as in the low-level rings, but it will allow full composability, for instance defining a polynomial with polynomial coefficients. Of course the `Poly` type also contains extended functionality that are not applicable in general rings, e.g., for accessing coefficients,.

Implementations of polynomial rings include dense, sparse, and functional representations (i.e., black-box polynomials), as described in the previous chapter. It should be noted that the black-box representation has no problem with basic arithmetic functions such as multiplication and division, but other functions such as retrieving the coefficient of a given monomial are not supported and will throw an error.

The `poly` class contains extensive implementations of efficient polynomial arithmetic, and in particular multiplication. Most of these algorithms are specialized for certain polynomial representations and in some cases certain coefficient rings as well, for maximum performance.

`alg`**:** This module contains will contain algorithms for higher-level computations such as factorization, decomposition, and interpolation. Current functionality is limited to the problems discussed in this thesis, but extending this should be quite easy. The hope is that this module will become a testbed for new algorithmic ideas in polynomial computations.

`test`**:** Correctness tests are contained in this module. The goal of these tests is not to prove correctness but rather to catch programming bugs. As with the rest of the library, the correctness tests are generic and in general consist of constructing random examples and testing whether ring identities hold. For instance, in testing rings, for many triples of randomly-chosen elements (a,b,c) from the ring, we confirm distributivity: that $a\cdot(b+c)=a\cdot b+a\cdot c$. This kind of correctness test would be easy to fool with an intentionally incorrect and dishonest implementation, but is quite useful in catching bugs in programming. Generic tests also have the significant advantage of making it easy to add new rings without writing entirely new testing suites.

`bench`**:** One of the main purposes in writing the MVP library is to compare the practical performance of algorithms. Hence accurate and fair benchmarking is extremely important.

Benchmarking, like correctness testing, is also handled generically, but in a slightly different fashion as it is important to achieve the highest possible performance.

Rather than compiling a single program to benchmark every implementation of a given data structure or algorithm, we generate two programs for each test case. One "dry run" initializes all the variables and in most cases makes two passes through the computation that is to be tested. The "actual run" program does the same but makes one more pass through the computation. In examining the difference between the performance of these two programs, we see the true cost of the algorithm in question, subtracting any initialization and memory overhead for initialisation and (de)allocation. This also allows the use of excellent tools such as `valgrind` to evaluate performance which can only be run externally on entire programs. Furthermore, the code generation, compilation, and timing is all handled automatically by scripts, requiring only a single generic `C++` program, and a simple text file indicating the macro definitions to substitute in for each test case.

We have thus far used the benchmarking routines to manually tune performance, choosing optimal crossover points between algorithms for our target architecture. In the future, this process could be automated, allowing greater flexibility and performance on a wide variety of machines.

2.4 Algorithms for low-level operations

The MVP library is designed to allow other packages to be plugged in for low-level arithmetic. For instance, adaptor classes exist to use some of NTL's types for modular and polynomial rings. However, the highest levels of performance for the most important operations are achieved with a bottom-up design and strong coupling between representations at all levels. Furthermore, at least including core functionality without requiring too many external packages will lead to more maintainable and (hopefully) longer-lasting code.

With this in mind, we now turn to the primitive operation of polynomial multiplication in $\mathbb{F}_p[x]$ for primes p that can be stored in a single machine word. This operation forms the basis for a variety of other algorithms, including long integer multiplication and representations of prime power fields. Furthermore, univariate polynomials of this kind will be used extensively as a test case for the algorithms developed in this thesis, as this domain bears true the often-useful algorithmic assumption that coefficients can be stored in a constant amount of space, and coefficient calculations take constant time.

We first examine algorithms and implementations for the coefficient field $\mathbb{F}_p$, and then turn to univariate multiplication over that field.

2.4.1 Small prime fields

Let p be a machine word-sized prime. Using the terminology of the IMM model, this means $p < 2^w$ for whatever problem size is under consideration. On modern hardware, this typically means p has at most 32 or 64 bits in the binary representation. For this discussion, we

assume that arithmetic with powers of 2 is highly efficient. This is of course true on modern hardware; multiplications, divisions, additions, and subtractions modulo a power of two can be performed extremely quickly with shifts and bitwise logical instructions.

Most of the content here is well-known and used in many existing implementations. Further references and details can be found in the excellent books by Brent and Zimmermann (2010) and Hankerson, Menezes, and Vanstone (2004).

Addition and subtraction

The standard method for arithmetic modulo p is simply to perform computations over $\mathbb{Z}$, reducing each result modulo p via an integer division with remainder. This integer remainder operation, given by the `%` operator in `C++`, is very expensive compared to other basic instructions, and the main goal of the algorithms we now mention is to avoid its use.

We immediately see how this can be done for the operations of addition of subtraction. Let $a, b \in \mathbb{F}_p$, so that $0 \leq a, b < p$ under the standard representation. Then $0 \leq a + b < 2p - 1$, so after computing $a + b$ over $\mathbb{Z}$, we simply check if the sum is at least p, and if so subtract p, reducing into the desired range. We call this check-and-subtract a "correction". Subtraction reduces to addition as well since $-a \in \mathbb{F}_p$ is simply $p - a$, which always falls in the proper range when a is nonzero.

Some modern processors have very long pipelines and use branch prediction to "guess" which way an upcoming `if` statement will be evaluated. A wrong guess breaks the pipeline and is very expensive. It turns out that, on such processors, the idea above is quite inefficient since it continually requires checking whether the result is at least p. Furthermore, this branch will be impossible to reliably predict since it will be taken exactly one half of the time (when the operands are uniformly distributed).

Victor Shoup's NTL includes a very clever trick for avoiding this problem. First, the sum (or difference) is put into the range $-p + 1 \leq s \leq p - 1$, so that we need to check whether s is negative, and if so, add p to it. The key observation is that an arithmetic right shift of s to full width (either 32 or 64 bits) results in either a word with all `1` bits, if s is negative, or all `0` bits, if s is positive. By computing a bitwise `AND` operation with p, we get either p, if s is negative, or 0 otherwise. Finally, this is added to s to normalize into the range $[0, p - 1]$. Hence a branch is avoided at the cost of doing a single right shift and bitwise `AND` operation.

Despite these tricks, the modular reduction following additions is still costly. If the prime modulus p is much smaller than word-sized, further improvements can be gained by delaying modular reductions in certain algorithms. For instance, Monagan (1993) demonstrated improvements using this idea in polynomial multiplication. Say p has ℓ fewer bits than the width of machine words, i.e., $2^\ell p < 2^w$. Then 2^ℓ integers mod p can be summed in a single machine word, and the remainder of this sum modulo p can be computed with just ℓ "correction" steps, either using the branching or arithmetic right shift trick as discussed above.

Barrett's algorithm for multiplication

We now turn to the problem of multiplication of two elements $a, b \in \mathbb{F}_p$. The algorithm of Barrett (1987) works for any modulus p and takes advantage of the fact that arithmetic with powers of 2 is very efficient. The idea is to precompute an approximation for the inverse of p, which will enable a close approximation to the quotient when ab is divided by p.

Suppose p has k bits, i.e., $2^{k-1} \leq p < 2^k$. Let $\mu = \lfloor 2^{2k}/p \rfloor$, stored as a precomputation. The algorithm to compute $ab \bmod p$ then proceeds in four steps:

1. $c_1 \leftarrow \lfloor ab/2^k \rfloor$
2. $q \leftarrow \lfloor c_1 \mu / 2^k \rfloor$
3. $r \leftarrow ab - qp$
4. Subtract p from r repeatedly until $0 \leq r < p$

If the algorithm is implemented over the integers as stated, we know that the value of r computed on Step 3 is less than $4p$, and hence Step 4 iterates at most 3 times. Thus the reduced product is computed using three multiplications as well as some subtractions and bitwise shift operations.

If p can be represented exactly as a single- or double-precision floating point number, then we can precompute $\mu = 1/p$ as a floating-point approximation and perform the computation on Steps 1 and 2 in floating point as well, without any division by powers of 2. This will guarantee the initial value of r to satisfy $-p < r < 2p$, so only 2 "corrections" on Step 4 are required. This is in fact the approach used in NTL library on some modern processors that feature very fast double-precision floating point arithmetic. The general idea of using floating-point arithmetic for modular computations has been shown quite useful, especially when existing numerical libraries can be employed (Dumas, Giorgi, and Pernet, 2008, e.g.,).

Montgomery's algorithm

The algorithm of Montgomery (1985) is similar to Barrett's and again requires precomputation with p, but gains further efficiency by changing the representation. This time, let $k \in \mathbb{N}$ such that $p < 2^k$, but where 2^k can be much larger than p (unlike Barrett's algorithm). The finite field element $a \in \mathbb{F}_p$ is represented by the product $2^k a \bmod p$. Notice that the standard algorithms for addition and subtraction can still be used, since $2^k a + 2^k b = 2^k (a+b)$.

Now write $s = 2^k a$ and $t = 2^k b$, and suppose we want to compute the product of a and b, which will be stored as $2^k ab \bmod p$, or $st/2^k \bmod p$. The precomputation for this algorithm is $\mu = -1/p \pmod{2^k}$, which can be computed using the extended Euclidean algorithm. Notice this requires p to be a unit in $\mathbb{Z}/2^k\mathbb{Z}$, which means p must be odd for Montgomery's algorithm.

Given μ, s, t, the product $st/2^k \bmod p$ is computed in three steps:

1. $q \leftarrow (st) \cdot \mu \mod 2^k$

2. $r \leftarrow (st + qp)/2^k$

3. If $r < p$ return r, else return $r - p$

As with Barrett's algorithm, the product is computed using only three integer multiplications. Montgomery's form is more efficient, however, since the product on Step 1 is a "short product" where only the low-order bits are needed, and there are fewer additions, subtractions, and shifts required. Of course, there is a higher overhead associated with converting to/from the Montgomery representation, but this is negligible for most applications.

We have found a few more optimisations to be quite useful in the Montgomery representation, some of which are alluded to by Li, Maza, and Schost (2009b). First, the parameter k should be chosen to align with the machine word size, i.e., $k = w$, typically either 32 or 64. Using this, arithmetic modulo 2^k is of course just single-precision arithmetic; no additional steps are needed for example to perform the computation in Step 1 above.

We additionally observe that the high-order bits of a single-precision product are often computed "for free" by machine-level `MUL` instructions. These bits correspond to quotients modulo 2^k, as is needed on Step 2 above. The high-order bits are not directly available in high-level programming languages such as `C++`, but by writing very small inline assembly loops they may be extracted. For instance, in the `x86_64` architecture of popular contemporary 64-bit processors, any word-sized integer multiplication stores the high-order bits of the product into the `EDX` register. We use this fact in our implementation of the Montgomery representation in the MVP library, with positive results.

The Montgomery representation also lends itself well to delayed modular reduction in additions and subtractions, as discussed above. Again, say p has ℓ fewer bits than the machine word size, so that $2^\ell p < 2^w$. Now consider two un-normalized integers s, t in the Montgomery representation, satisfying $0 \le s, t \le 2^{\lfloor \ell/2 \rfloor} p$. The result r computed as their product in Step 2 of the Montgomery multiplication algorithm above is then bounded by

$$r = \frac{st + qp}{2^w} < \frac{2^\ell p^2 + 2^w p}{2^w} < 2p.$$

Hence $2^{\lfloor \ell/2 \rfloor}$ elements in $\mathbb{F}_p$ can be safely summed without performing any extra normalization.

In the MVP library, we embrace this by always storing finite field elements in the Montgomery reduction redundantly, as integers in the range $\{0, 1, \ldots, 2p\}$. When the modulus p is not too large, this allows direct remainder operations or "normalizations" to be almost entirely avoided in arithmetic calculations. The cost of comparisons is slightly increased, making this representation more useful for dense polynomial arithmetic, as opposed to sparse methods which frequently test for zero. Observe that zero testing is especially bad because there are three possible representations of zero: 0, p, or $2p$.

2.4.2 Univariate polynomial multiplication

The MVP library contains dense univariate polynomial multiplication routines incorporating the quadratic "school" method, Karatsuba's divide-and-conquer method, as well as FFT-

based multiplication. The choice of algorithm for different ranges of input size is based on benchmarking on our test machine. We also implemented the well-known blocking algorithm for unbalanced polynomial multiplication, when the degrees of the two operands differ significantly. Roughly speaking, the idea is to break the larger polynomial into blocks equal in size to that of the smaller polynomial, compute the products separately, and then sum them. We will discuss all these algorithms for dense multiplication much more extensively in the next two chapters.

Multiplication of sparse univariate polynomials in the MVP library is performed using the algorithm of Johnson (1974), which has seen recent success in the work of Monagan and Pearce (2007, 2009). We briefly explain the idea of this algorithm. Given two polynomials $f, g \in \mathsf{R}[x]$ to be multiplied, write

$$f = a_1 x^{d_1} + a_2 x^{d_2} + \cdots + a_s x^{d_s}$$
$$g = b_1 x^{e_1} + b_2 x^{e_2} + \cdots + b_t x^{e_t}.$$

As usual assume $d_1 < d_2 < \cdots < d_s = \deg f$, and similarly for the e_i's.

The terms in the product $f \cdot g \in \mathsf{R}[x]$ are computed one term at a time, in order. To accomplish this, we maintain a min-heap of size s which initially contains the pairs (d_i, e_1) for $1 \leq i \leq s$. The min-heap is ordered on the sum of the two exponents in each element, which corresponds to the degree of the single term in $f \cdot g$ that is the product of the corresponding terms in f and g. (For a refresher on heaps and other basic data structures, see any algorithms textbook such as Cormen, Leiserson, Rivest, and Stein (2001).)

From this ordering, the top of the min-heap always represents the next term (in increasing degree order) in the polynomial product we are computing. After adding the next term to a running total for $f \cdot g$, we update the corresponding pair in the heap by incrementing the second exponent along the terms in g. So, for example, the pair (d_3, e_5) is updated to (d_3, e_6), and then the heap is updated to maintain min-heap order, using $d_3 + e_6$ for this updated element's key. As the second exponent in each pair goes past b_t, it is removed from the heap, until the heap is empty and we are done.

For the cost analysis, notice that the number of times we must examine the top of the heap and update it is exactly $s \cdot t$, the total number of possible terms in the product. Because the heap has size s, the total time cost is $O(s\,t \log s)$ exponent operations and $O(s\,t)$ ring operations. In the IMM model, assuming each exponent requires at most k words of storage, this gives a total cost of $O(k\,s\,t \log s)$ word operations. Observe that k is proportional to $\log \deg(f \cdot g)$.

This time cost matches other previously-known algorithms, for instance the "geobucket" data structure (Yan, 1998). However, the storage cost for this heap-based algorithm is better — only $O(s)$ words of intermediate storage are required, besides the input and output storage. In addition, since s and t are known, we can assume without loss of generality that $s \leq t$, improving the cost in unbalanced instances.

System component	Version / Capacity
Processor	AMD Phenom II
Processor speed	3.2 GHz
L1 Cache	512 KiB
L2 Cache	2 MiB
L3 Cache	8 MiB
Main memory (RAM)	8 GiB
Linux kernel	2.6.32-28-generic

Table 2.1: Properties of machine used for benchmarking

2.5 Benchmarking of low-level operations

We now compare our implementations of two low-level algorithms described above against the state of the art in existing software. The goal is not to claim that we have the best implementation of every algorithm, but merely that our algorithms are comparable to existing fast implementations. As these low-level routines are used in many of the higher-level algorithms discussed later in this thesis, the results here provide a justification for developing such algorithms in the MVP library.

All the timings given are on the 64-bit machine described in Table 2.1. We also used this machine to benchmark all the other implementations discussed in later chapters.

First we examine algorithms for finite field arithmetic. The MVP library currently contains a few implementations of finite field arithmetic for various purposes, but the fastest, at least for dense algorithms, is the Montgomery representation described above. We compared this representation against NTL's `zz_p` type for integers modulo a single-precision integer.

For the benchmarking, we used the so-called "axpy" operation that is a basic low-level operation for numerous algorithms in polynomial arithmetic as well as linear algebra. In this context, the axpy operation simply means adding a product ax to an accumulator y. Hence each axpy operation requires a single multiplication and addition in the ring.

NTL `zz_p`	MVP `ModMont`	Ratio
130.07 s	89.66 s	.689

Table 2.2: Benchmarking for ten billion axpy operations modulo a word-sized prime

Table 2.2 compares the cost, in CPU seconds, of 10^{10} (that is, ten billion) consecutive axpy operations using MVP's `ModMont` class (Montgomery representation), and using NTL's `zz_p`. These were compared directly in the `bench` module of MVP, using the "dry run" and actual run method described earlier to get accurate and meaningful results. We have also listed the ratio of time in MVP divided by time in NTL. So a lower number (and, in particular, a value less than one) means that our methods are outperforming NTL. This model will be repeated

Degree	Sparsity	SDMP time	MVP time	Ratio
1 000	100	9.25×10^{-4}	5.34×10^{-4}	.577
10 000	100	1.15×10^{-3}	6.47×10^{-4}	.563
1 000 000	100	1.21×10^{-3}	6.81×10^{-4}	.563
1 000	500	1.80×10^{-2}	1.03×10^{-2}	.572
10 000	500	2.58×10^{-2}	1.77×10^{-2}	.686
1 000 000	500	3.75×10^{-2}	2.08×10^{-2}	.555
1 000	1 000	6.23×10^{-2}	2.63×10^{-2}	.422
10 000	1 000	9.49×10^{-2}	6.99×10^{-2}	.737
1 000 000	1 000	1.62×10^{-1}	9.32×10^{-2}	.575
10 000	2 000	.337	.274	.813
1 000 000	2 000	.624	.427	.684
10 000	5 000	1.82	1.36	.747
1 000 000	5 000	3.53	3.20	.907

Table 2.3: Benchmarks for sparse univariate multiplication in MVP and SDMP

for most timing results we give, always with the time for the method under consideration in the numerator, and the time for the "standard" we are comparing against in the denominator.

The second problem we benchmarked is univariate sparse polynomial multiplication. Since multivariate sparse polynomials are represented in univariate using the Kronecker substitution, the univariate multiplication algorithm is of key importance. In these benchmarks, the coefficient ring is $\mathbb{F}_p$ for a single-precision integer p. Interestingly, we found that the cost was nearly the same using our Montgomery representation or NTL's `zz_p` representation. This is likely due in part to the fact that ring operations do not dominate the cost of sparse multiplication, as well as the increased cost of equality testing in our redundant Montgomery representation.

For comparison, we used the SDMP package for sparse polynomial arithmetic by Michael Monagan and Roman Pearce that has been integrated into MAPLE 14. Benchmarking our `C++` program against a general-purpose computer algebra system with a run-time interpreted language such as MAPLE is inherently somewhat problematic. However, we tried to be as fair as possible to SDMP. We computed the timing difference between a "dry run" and an actual run, just as with the other benchmarking in MVP. We found that MAPLE spends a lot of time doing modular reductions, and so just computed the product over the integers in MAPLE, although our code took the extra time to do the modular reductions. Finally, we used the `SDMPolynom` routine in MAPLE to convert to the SDMP representation, and did not write out or echo any results. Since the SDMP library itself is (to our knowledge) written in `C` code linked with the MAPLE system, we feel this gives a very fair comparison to the MVP software.

The results of the sparse univariate multiplication benchmarks are shown in Table 2.3. As we can see, benchmarking sparse polynomial operations involves more parameters than just the number of iterations. The times shown are for a single multiplication with the specified degree and sparsity in each system, and in every test case both polynomials were randomly generated using the same parameters. In the actual benchmarking, we increased the number

of iterations appropriately to get at least four significant figures of timing information for each example (on our machine, this meant at least ten seconds). As always, the ratio is the time of our implementation divided by the standard, which in this case is SDMP. Inverting these, we see that MVP is consistently between once and twice as fast as SDMP.

Conspicuously absent from this section is any discussion of dense polynomial arithmetic. While this is a primitive operation at least as important as those discussed here, we will explore this topic more thoroughly in Chapter 4. The new algorithms we present there have been implemented in the MVP library, and we show benchmarking comparisons against existing software in that chapter.

2.6 Further developments

The MVP library now contains fast implementations for basic arithmetic in a variety of polynomial rings, sparse interpolation over certain domains, and a few algebraic algorithms for sparse polynomials that will be presented later in this thesis. However, the software is still quite young and there is much work to be done before it will be useful to a wide audience. We mention now some of the most important tasks to improve the MVP library's usefulness.

First, a greater variety of algebraic algorithms should be included for basic problems such as factorization. One of our chief motivations in developing a new library was to give a good implementation of the sparse factorization algorithms by Lenstra (1999); Kaltofen and Koiran (2005, 2006), and this is certainly a key priority. Of course, dense factorization methods are also important and should be included. For this, it would be useful to tie in with existing fast dense multiplication algorithms implemented in NTL and FLINT.

Another major and important task for the future development of MVP is to build in parallelization. This will of course require not just careful implementations but in many cases new algorithmic ideas for a number of important, basic problems. Allowing for parallelization without affecting the sequential performance of algorithms will also be an important consideration and software engineering challenge. We have tried to design the library with thread safety in mind, but in some cases using global or static variables can drastically improve performance. Balancing this with the needs of parallel computation will present difficult and interesting challenges.

Finally, the success and longevity of any software package is heavily dependent on building and maintaining a community of users and developers. While the library has been solo-authored to begin with, it is hoped that more researchers and students will get involved with developing the library along the lines mentioned above. Ultimately, we would like MVP to be a popular venue to try and compare implementations of new algorithms for polynomial computations. In addition, if the library is fast and solves a variety of problems, it may be useful as an add-on package to larger systems such as SAGE or SINGULAR, further increasing the visibility and user base.

Bibliography

John Abbott and Anna Maria Bigatti. CoCoALib: a C++ library for doing computations in commutative algebra. Online, February 2011.
URL http://cocoa.dima.unige.it/cocoalib. Version 0.9942. Referenced on page 22.

Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In Andrew Odlyzko, editor, *Advances in Cryptology âĂŤ CRYPTOâĂŹ 86*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer Berlin / Heidelberg, 1987.
doi: 10.1007/3-540-47721-7_24. Referenced on page 29.

Michael Ben-Or and Prasoon Tiwari. A deterministic algorithm for sparse multivariate polynomial interpolation. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, STOC '88, pages 301–309, New York, NY, USA, 1988. ACM. ISBN 0-89791-264-0.
doi: 10.1145/62212.62241. Referenced on page 24.

Richard Brent and Paul Zimmermann. *Modern Computer Arithmetic*. Number 18 in Cambridge Monographs on Applied and Computational Mathematics. Cambridge Univ. Press, November 2010. Referenced on page 28.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, second edition, September 2001. ISBN 0262032937. Referenced on page 31.

W. Decker, G.-M. Greuel, G. Pfister, and H. Schönemann. SINGULAR 3-1-2 — A computer algebra system for polynomial computations. Online, 2010.
URL http://www.singular.uni-kl.de. Referenced on page 22.

Angel Díaz and Erich Kaltofen. FOXBOX: a system for manipulating symbolic objects in black box representation. In *Proceedings of the 1998 international symposium on Symbolic and algebraic computation*, ISSAC '98, pages 30–37, New York, NY, USA, 1998. ACM. ISBN 1-58113-002-3.
doi: 10.1145/281508.281538. Referenced on page 23.

J.-G. Dumas, T. Gautier, M. Giesbrecht, P. Giorgi, B. Hovinen, E. Kaltofen, B. D. Saunders, W. J. Turner, and G. Villard. LINBOX: A generic library for exact linear algebra. In Arjeh M Cohen, Xiao-Shan Gao, and Nobuki Takayama, editors, *Mathematical software*, Proc. First International Congress of Mathematical Software, pages 40–50. World Scientific, 2002.
doi: 10.1142/9789812777171_0005. Referenced on page 24.

Jean-Guillaume Dumas, Pascal Giorgi, and Clément Pernet. Dense linear algebra over word-size prime fields: the FFLAS and FFPACK packages. *ACM Trans. Math. Softw.*, 35:19:1–19:42, October 2008. ISSN 0098-3500.
doi: 10.1145/1391989.1391992. Referenced on page 29.

Timothy S. Freeman, Gregory M. Imirzian, Erich Kaltofen, and Lakshman Yagati. Dagwood: A system for manipulating polynomials given by straight-line programs. *ACM Trans. Math. Softw.*, 14:218–240, September 1988. ISSN 0098-3500.
doi: 10.1145/44128.214376. Referenced on page 23.

Mickaël Gastineau and Jacques Laskar. TRIP: A computer algebra system dedicated to celestial mechanics and perturbation series. *SIGSAM Bull.*, 44:194–197, January 2011. ISSN 0163-5824.
doi: 10.1145/1940475.1940518. Referenced on page 22.

Torbjörn Granlund et al. *GNU Multiple Precision Arithmetic Library, The*. Free Software Foundation, Inc., 4.3.2 edition, January 2010.
URL http://gmplib.org/. Referenced on page 22.

Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide to elliptic curve cryptography*, chapter 2: Finite Field Arithmetic, pages 25–74. Springer Professional Computing. Springer-Verlag, New York, 2004. Referenced on page 28.

William Hart, Fredrick Johansson, and Sebastian Pancratz. *FLINT: Fast Library for Number Theory*, version 2.0.0 edition, January 2011.
URL http://www.flintlib.org/. Referenced on page 22.

David Harvey. zn_poly: a C library for polynomial arithmetic in $\mathbb{Z}/n\mathbb{Z}[x]$. Online, October 2008.
URL http://cims.nyu.edu/~harvey/code/zn_poly/. Version 0.9. Referenced on page 22.

Stephen C. Johnson. Sparse polynomial arithmetic. *SIGSAM Bull.*, 8:63–71, August 1974. ISSN 0163-5824.
doi: 10.1145/1086837.1086847. Referenced on page 31.

Erich Kaltofen and Pascal Koiran. On the complexity of factoring bivariate supersparse (lacunary) polynomials. In *ISSAC '05: Proceedings of the 2005 international symposium on Symbolic and algebraic computation*, pages 208–215, New York, NY, USA, 2005. ACM. ISBN 1-59593-095-7.
doi: 10.1145/1073884.1073914. Referenced on page 34.

Erich Kaltofen and Pascal Koiran. Finding small degree factors of multivariate supersparse (lacunary) polynomials over algebraic number fields. In *ISSAC '06: Proceedings of the 2006 international symposium on Symbolic and algebraic computation*, pages 162–168, New York, NY, USA, 2006. ACM. ISBN 1-59593-276-3.
doi: 10.1145/1145768.1145798. Referenced on page 34.

H. W. Lenstra, Jr. Finding small degree factors of lacunary polynomials. In *Number theory in progress, Vol. 1 (Zakopane-Kościelisko, 1997)*, pages 267–276. de Gruyter, Berlin, 1999. Referenced on pages 24 and 34.

Xin Li, Marc Moreno Maza, Raqeeb Rasheed, and Éric Schost. The modpn library: Bringing fast polynomial arithmetic into MAPLE. *ACM Commun. Comput. Algebra*, 42:172–174, February 2009a. ISSN 1932-2240.
doi: 10.1145/1504347.1504374. Referenced on page 23.

Xin Li, Marc Moreno Maza, and Éric Schost. Fast arithmetic for triangular sets: From theory to practice. *Journal of Symbolic Computation*, 44(7):891–907, 2009b. ISSN 0747-7171.
doi: 10.1016/j.jsc.2008.04.019. International Symposium on Symbolic and Algebraic Computation. Referenced on page 30.

Michael Monagan. In-place arithmetic for polynomials over z_n. In John Fitch, editor, *Design and Implementation of Symbolic Computation Systems*, volume 721 of *Lecture Notes in Computer Science*, pages 22–34. Springer Berlin / Heidelberg, 1993.
doi: 10.1007/3-540-57272-4_21. Referenced on page 28.

Michael Monagan and Roman Pearce. Polynomial division using dynamic arrays, heaps, and packed exponent vectors. In Victor Ganzha, Ernst Mayr, and Evgenii Vorozhtsov, editors, *Computer Algebra in Scientific Computing*, volume 4770 of *Lecture Notes in Computer Science*, pages 295–315. Springer Berlin / Heidelberg, 2007.
doi: 10.1007/978-3-540-75187-8_23. Referenced on page 31.

Michael Monagan and Roman Pearce. Parallel sparse polynomial multiplication using heaps. In *Proceedings of the 2009 international symposium on Symbolic and algebraic computation*, ISSAC '09, pages 263–270, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-609-0.
doi: 10.1145/1576702.1576739. Referenced on pages 23 and 31.

Michael Monagan and Roman Pearce. Sparse polynomial division using a heap. *Journal of Symbolic Computation*, In Press, Corrected Proof, 2010a. ISSN 0747-7171.
doi: 10.1016/j.jsc.2010.08.014. Referenced on page 23.

Michael Monagan and Roman Pearce. Parallel sparse polynomial division using heaps. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, PASCO '10, pages 105–111, New York, NY, USA, 2010b. ACM. ISBN 978-1-4503-0067-4.
doi: 10.1145/1837210.1837227. Referenced on page 23.

Peter L. Montgomery. Modular multiplication without trial division. *Math. Comp.*, 44(170): 519–521, 1985.
doi: 10.2307/2007970. Referenced on page 29.

PARI/GP, version 2.3.5. The PARI Group, Bordeaux, February 2010.
URL http://pari.math.u-bordeaux.fr/. Referenced on page 22.

Victor Shoup. NTL: A Library for doing Number Theory. Online, August 2009.
URL http://www.shop.net/ntl/. Version 5.5.2. Referenced on page 22.

Thomas Yan. The geobucket data structure for polynomials. *Journal of Symbolic Computation*, 25(3):285–293, 1998. ISSN 0747-7171.
doi: 10.1006/jsco.1997.0176. Referenced on page 31.